CS 423
Operating System Design: The Linux Scheduler

Professor Adam Bates
Spring 2017
Goals for Today

• **Learning Objective:**
  • Understand high-level view of Linux scheduling evolution

• **Announcements, etc:**
  • MP1 is out! Due **Feb 15**.

• **Pre-Lecture Discussion:**
  • Explaining the “Processing Hardware Interrupts” chart
  • Homework 1 wrap-up

**Reminder:** Please put away devices at the start of class
Device controller or other hardware issues an interrupt.

Processor finishes execution of current instruction.

Processor signals acknowledgment of interrupt.

Processor pushes PSW and PC onto control stack.

Processor loads new PC value based on interrupt.

Save remainder of state information???

Process interrupt.

Restore process state information.

Restore old PSW and PC.
HW1 Wrap-Up

• Average Score: 88.7%…

• Three questions of medium difficulty (<80%):
  • “Hold-and-wait” situations?
  • which code snippets are wrong?
  • UNIX shell?
Which of the following systems may never exhibit a "hold-and-wait" situation? You may assume that the only blocking that occurs in these systems occurs on mutexes.

a. Systems that ensure that all resources needed for an application are locked in a single atomic operation that either succeeds and locks all requested resources or fails and locks none.

b. Systems with only one mutex.

c. Systems where all mutexes are numbered. A user cannot lock a mutex with a lower number, $X$, after they have locked a mutex with a larger number, $Y > X$.

d. Systems of type (a) and (b) only

e. Systems of type (a), (b), or (c)
Which of the following code snippets are wrong?

Case 1
int *p;
*p=10;

Case 2
char a[2];
strcpy (a, “Hi”);

Case 3
int b[10];
*b=11;
Which of the following best describes a UNIX shell?

a. Part of the UNIX kernel that executes user commands
b. A process forked off at UNIX initialization to accept inputs from a user
c. A system call executed by the UNIX startup routine to accept commands from users
d. A library that implements various UNIX commands
e. The UNIX keyboard device driver that interprets keyboard input
What Are Scheduling Goals?

• What are the goals of a scheduler?

• Linux Scheduler’s Goals:
  ▪ Generate illusion of concurrency
  ▪ Maximize resource utilization (e.g., mix CPU and I/O bound processes appropriately)
  ▪ Meet needs of both I/O-bound and CPU-bound processes
    ▪ Give I/O-bound processes better interactive response
    ▪ Do not starve CPU-bound processes
  ▪ Support Real-Time (RT) applications
Early Linux Schedulers

- **Linux 1.2**: circular queue w/ round-robin policy.
  - Simple and minimal.
  - Did not meet many of the aforementioned goals

- **Linux 2.2**: introduced scheduling classes (real-time, non-real-time).
2.4: O(N) scheduler.

- Epochs → slices: when blocked before the slice ends, half of the remaining slice is added in the next epoch.
- Simple.
- Lacked scalability.
- Weak for real-time systems.
Linux 2.6 Scheduler

- O(1) scheduler
- Tasks are indexed according to their priority [0,139]
  - Real-time [0, 99]
  - Non-real-time [100, 139]
## Scheduling API

<table>
<thead>
<tr>
<th>System call</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nice()</td>
<td>change the priority</td>
</tr>
<tr>
<td>getpriority()</td>
<td>get the maximum group priority</td>
</tr>
<tr>
<td>setpriority()</td>
<td>set the group priority</td>
</tr>
<tr>
<td>sched_getscheduler()</td>
<td>get the scheduling policy</td>
</tr>
<tr>
<td>sched_setscheduler()</td>
<td>set the scheduling policy and priority</td>
</tr>
<tr>
<td>sched_getparam()</td>
<td>get the priority</td>
</tr>
<tr>
<td>sched_setparam()</td>
<td>set the priority</td>
</tr>
<tr>
<td>sched_yield()</td>
<td>relinquish the processor voluntarily</td>
</tr>
<tr>
<td>sched_get_priority_min()</td>
<td>get the minimum priority value</td>
</tr>
<tr>
<td>sched_get_priority_max()</td>
<td>get the maximum priority value</td>
</tr>
<tr>
<td>sched_get_priority_max()</td>
<td>get the maximum priority value</td>
</tr>
<tr>
<td>sched_rr_get_interval()</td>
<td>get the time quantum for Round-Robin</td>
</tr>
</tbody>
</table>
Resource Sharing

Two Fundamental Mechanisms...

- Prioritization
- Resource partitioning
Prioritization

SCHED_FIFO

- Used for real-time processes
- Conventional preemptive fixed-priority scheduling
  - Current process continues to run until it ends or a higher-priority real-time process becomes runnable
- Same-priority processes are scheduled FIFO
Partitioning

- Used for real-time processes
- CPU “partitioning” among same priority processes
  - Current process continues to run until it ends or its time quantum expires
  - Quantum size determines the CPU share
- Processes of a lower priority run when no processes of a higher priority are present
SCHED_NORMAL

- Used for non real-time processes
- Complex heuristic to balance the needs of I/O and CPU centric applications
- Processes start at 120 by default
  - **Static priority**
    - A “nice” value: 19 to -20.
    - Inherited from the parent process
    - Altered by user (negative values require special permission)
  - **Dynamic priority**
    - Based on static priority and applications characteristics (interactive or CPU-bound)
    - Favor interactive applications over CPU-bound ones
- Timeslice is mapped from priority
Coming up…

- Linux 2.6.23 and beyond (CFS)
- Offers a specified fair share of the CPU to each process:
  - A process who is furthest behind its share gets to run next